Flush+Reload between Docker Containers

Samuil Orlioglu
sorliog@clemson.edu
Clemson University
Clemson, SC, USA

Anvitha Yerneni
ayernen@clemson.edu
Clemson University
Clemson, SC, USA

Abstract

Side-channel and micro-architectural attacks have been of heightened interest in latest years. With the rise of cloud computing, we see that there is an increase of users deploying their application into the cloud. Cloud providers such as AWS use containerization technology to serve these servlerless applications. We set out to investigate the accuracy of the Flush+Reload side channel attack in a constructed container environment with Docker. We discover that Flush+Reload is not effective on containers running on a virtual machine. Despite this, we show that Flush+Reload 83.3% effective against containers running on OS installed on a bare-metal machine.

Keywords: cache, side-channel, container, virtual machine, assembly

1 Introduction

With the adoption of computer systems in world, computer security has been rising as a research interest. Many security threats try to penetrate a secured system directly. In recent years, researchers have taken interest in a class of attacks that are stealthier than traditional attacks. This class of attacks, dubbed side-channel attacks exploit meta-information of the system's operations. Cache side-channel attacks exploit CPU cache to extract or even disrupt the computer's operations. Flush+Reload[6] is one such attack that can extract the information about the processes running on a system. In this project, we investigate the efficacy of Flush+Reload between process running on separate Docker containers.

2 Background

Modern CPU processors have incorporated cache memory to assist in retrieving frequently accessed memory pages. Cached data allows the CPU to avoid fetching data directly from main memory. This is a benefit because directly accessing the main memory is a relatively slow process and inevitably creates CPU idle time. Most computer architecture have a CPU with multiple cores. Most of these systems have a 3-level cache system in which cache level one and two are not shared between cores. However, level three cache (also known as last level cache) is commonly shared between all cores.

Recent trends in CPU micro-architecture design tend to non-inclusive caches to protect against cache security issues. However, even with non-inclusive cache, certain timing attacks are still possible, as we demonstrate in this project.

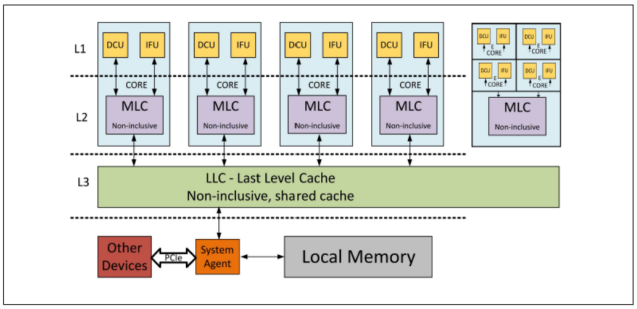

Figure 1. 12th Gen Intel CPU cache structure [2]

2.1 Flush+Reload

Flush+Reload is a side channel attack that allows an attacking process to spy and leak data from a victim process by exploiting the shared last level cache. Commonly, this attack monitors a shared memory address and it's usage in a concurrent victim process. Flush+Reload continually flushes the entire cache and waits for the victim process to access the shared memory address. Then, the attacker accesses the same shared memory address and measures the access time. If the access time is shorter than some specified threshold, then the victim has indeed accessed the shared library. And conversely, if the attacker's access time is longer than the threshold, then the victim process has not accessed the shared library.

2.2 Docker

Docker is a containerization technology that allows a user to create "containers": isolated virtualized environments that are used to run user's software. These containers can be deployed in the cloud as services for users. Containers are distinguished virtual machines in that containers share the host operating systems kernel, whereas, virtual machines incorporate their own kernel. Given that these containers run in parallel on an underlying physical system, it may be possible that a malicious actor can deploy a container with the purpose of spying on other containers running on the system. Since Flush+Reload attacks work even between virtual machines, it is logical to assume that Flush+Reload will work between containers, thereby posing a risk to many applications. Flush+Reload can be used to steal private keys, log network activity, work as a keylogger and many more malignant possibilities.

Since Docker is a containerization technology that is similar

the ones that cloud providers use, namely AWS' Firecracker [1], we can place-hold cloud containerization with Docker in this project.

2.3 Shared Memory and ASLR

In a modern computer system, virtual memory is used to abstract the physical memory to many processes. In many cases, multiple processes use the same library in their execution. One approach to save physical memory is where the OS stores a single instance of the shared library physically on the memory, but maps this location to address in the virtual space of the processes that require this library.
In modern Linux systems, a security feature called Address Space Layout Randomization (ASLR) protects against a number of memory attacks by offsetting import addresses by a random amount. In order to very that our processes use the same shared library address, we must first, necessarily turn off ASLR.

3 Threat Model and Lab environment

3.1 Threat model

We assume the existence of a victim container running a victim process concurrently with an attacker container running. The malicious actor has little control of the victim container, but has complete control over the attacker container. The victim container and the victim process are running on the same underlying machine as the attacker container. We assume that the victim process is executing some number of confidential function, such as encrypting data with a key, with the SquareMultiply algorithm.
The attacking process and victim process share a library. In order to avoid system level noise in our experiments, we create our own shared library with functions that simulate the Square and Multiply functions of the SquareMultiply algorithm. We turn off ASLR to verify that the processes are utilizing the same virtual memory address for the shared library. In theory, this attack would work even with ASLR turned on.

3.2 Lab environment

We run this attack in two configurations:

- on Docker containers running on a VM
- on Docker containers running on a physical computer

3.2.1 Virtual Machine setup. The virtual machine is an Ubuntu 20.04 Focal Fossa 64 bit running on Virtualbox. The underlying hardware is Windows 10 with Intel Core i5-1245U processor. This processor features 12 MB of last level non-inclusive cache [5].

3.2.2 Bare-metal setup. The bare-metal setup is a HP ProLiant ML10 v2 server with Ubuntu 20.04 Focal Fossa 64 bit running. The processor is Intel core i3-4150. This processor features 3 MB of last level non-inclusive cache [4].

4 Attack Structure

There are three parts to the attack: the shared library, victim code and attacking code.

4.1 Shared Library

The shared library has two functions: `square` and `multiply`.

```
int square(int value) {
    return value * value;
}

int multiply(int value, int factor) {
    return value * factor;
}
```

Listing 1. shared.cpp

The implementation of these functions is not within the scope of this project, rather the purpose is to track function calls to these functions.

4.2 Victim code

The victim code simulates encryption with the functions of the shared library. We also keep track of timestamp of each function call and print the name of the function called.

```
std::time_t time;
int m;
int i;
int j;

std::time_t start;
start = std::time(0);

while (std::time(0) - start <= 10) {
  // Wait some time
    usleep(600000);
    time = std::time(0);
    if (i % 2 == 1) {
        // If odd, then multiply
        i = multiply(i, j) % m;
        time = std::time(0);
        printf("[%ld] multiply %d\n", time, i);
    } else {
        // else square
        i = square(i) % m;
        time = std::time(0);
        printf("[%ld] square %d\n",time, i);
    }
}
```

Listing 2. victim.cpp

In order to establish consistency between trials, the loop is set to run for at most 10 seconds.

4.3 Attacker code

There are two attack configurations.

4.3.1 attacker.cpp.

The first file, `attacker.cpp` attaches itself to the address of the `multiply()` function. There two functions that allow the attack. The `flush()` function flushes (clears) the given address out of the cache. This is the flush step of Flush+Reload. It uses the inline assembly feature of C++ to call the `clflush` instruction.

The `check()` function checks if the address was cached by timing the access time. If the access time is less than the `threshold`, then we can assume that this address was cached. This is the reload step of Flush+Reload. We also accomplish this with the inline assembly feature of C++.

The `threshold` value is a user defined constant. It is carefully chosen such that the number of false positives is minimized and so that the number of false negatives are also minimized.

```
unsigned int check(const void *adrs,
const unsigned long threshold) {
    volatile unsigned long time;
    asm __volatile__ (
        "mfence                 \n"
        "lfence                 \n"
        "rdtsc                  \n"
        "movl %%eax, %%esi      \n"
        "movl (%1), %%eax       \n"
        "lfence                 \n"
        "rdtsc                  \n"
        "subl %%esi, %%eax      \n"
        "clflush 0(%1)          \n"
        : "=a" (time)
        : "c" (adrs)
        : "%esi", "%edx"
        );
    return time < threshold;
}


void flush(const void *adrs) {
    asm __volatile__ (
      "mfence             \n"
      "lfence             \n"
      "clflush 0(%0)      \n"::"c" (adrs)
        );
}
```

Listing 3. attacker.cpp

The driver code establishes the address of the `multiply()` function in the variable `addrs`. In an infinite loop, we flush the cache, sleep for a second, and check if the `addrs` was accessed.

```
void main(){
    void *addrs = (void *) multiply;
  while (1) {
        // Clear the cache
        flush(addrs);
        time = std::time(0);

        // Wait some time
        while (std::time(0) <= time){
            usleep(10);
        }


        // Check if addrs was accessed
        int accessed = check(addrs,
        threshold);
        time = std::time(0);
        printf("[%ld] Accessed addr
        multiply:%d\n",
         time, accessed);
    }

}
```

Listing 4. attacker.cpp

4.3.2 attacker2.cpp.

The second attack is similar to the first attack code, but in addition to flushing and reloading the address for `multiply()`, we also track `square()`

4.4 Attack execution

First, we create two Docker containers that share a volume that contains the code. We compile the files into executables. On the designated attacker container, we launch the attack with the desired `threshold` value.

While the attack is running, in the victim container, we launch the victim process. We observe the output of the attacker to see if the attack captures the victim processes function calls.

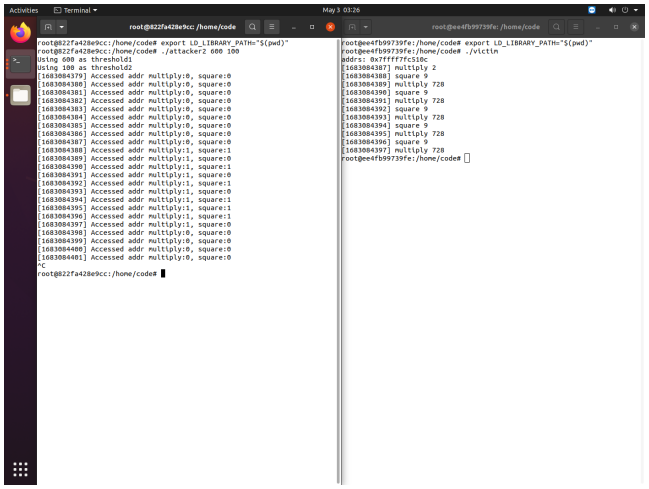

Figure 2. Executing the attack on the bare-metal setup. The attacker is the left terminal window and the victim is the right terminal window. Notice that as soon as the victim runs, the attacker side displays 1's.

4.5 Caveats

In order for the binaries to execute successfully, it is necessary to set the `LD_LIBRARY_PATH` environment variable to the directory where the compiled shared library is located.

5 Evaluation

We ran the attacks on the VM and the bare-metal setup. The goal was to find a threshold value for any particular trial such that the accuracy of the attack is maximized.
We define the accuracy as the ratio of correctly predicted function calls by the attacker to the total number of function calls by the victim. We include two seconds before and after the victim executed in the calculation for accuracy.

6 Results

The results show that when tracking on the `multiply()` function, the VM setup outperformed the bare-metal setup with an accuracy of ≈ 0.786. This is shown in Figure 3 and Figure 4.

Comparing accuracy at various thresholds on VM and Bare-metal setups

Using `attacker.cpp`

Threshold	Accuracy (VM)	Accuracy (Bare-metal)
700	0.714285714	0.571428571
800	**0.785714286**	0.642857143
900	0.642857143	0.642857143
1000	0.571428571	**0.714285714**
1100	0.428571429	0.642857143
1200	0.357142857	0.571428571

Figure 3. Executing `attacker.cpp` on the VM and bare-metal on various thresholds.

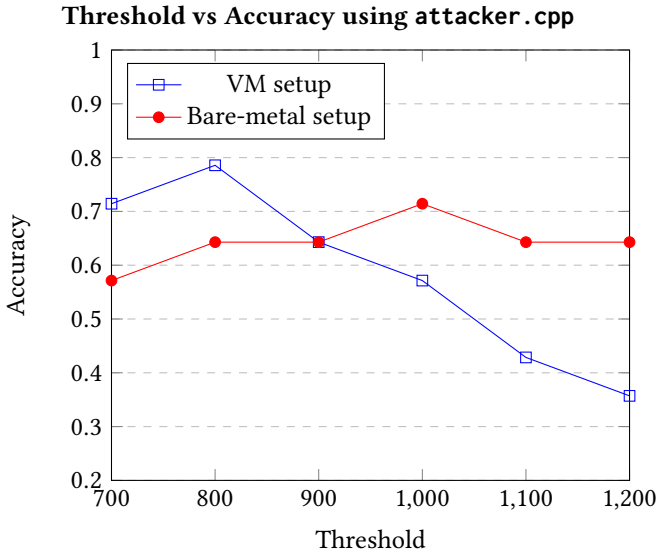

Figure 4. Threshold vs Accuracy using `attacker.cpp` on the VM configuration. Notice the peak accuracy of ≈ 0.786 at threshold of 800

Comparing accuracy at various thresholds on VM and Bare-metal setups, with both `multiply` and `square`

Using `attacker2.cpp`

Threshold (multiply)	VM (multiply)	Bare-metal (multiply)
500	0.595238095	0.595238095
600	0.547619048	0.642857143
700	**0.666666667**	0.619047619
Threshold (square)	**VM (square)**	**Bare-metal (square)**
50	0.642857143	0.738095238
100	0.595238095	**0.833333333**
150	0.547619048	0.761904762

Figure 5. Executing `attacker2.cpp` on the VM and bare-metal on various thresholds for both `square()` and `multiply()`.

However, when we also track the `square()` function in addition to the `multiply()` function, we find that the bare-metal setup outperforms the VM with an accuracy of ≈ 0.833 when tracking the `square()` function while the `square()` function on the VM setup decreases in accuracy. See Figure 5 and Figure 6.

6.1 Hypothesis

It is interesting that VM accuracy does not outperform the bare-metal accuracy. I hypothesize the the virtual machine hypervisor might be interfering with the shared address of the cache.
It is true that this attack assumes many factors that might not be in place. Despite this, the principle of the timing attack on

shared cache still poses a threat. Due to the way the Square-Multiply algorithm is structured, an attacker only needs to guess at least one function (either square xor multiply) with high accuracy to reconstruct the encryption key. Guessing the square function with 83.3% accuracy does in deed pose a threat to these systems.

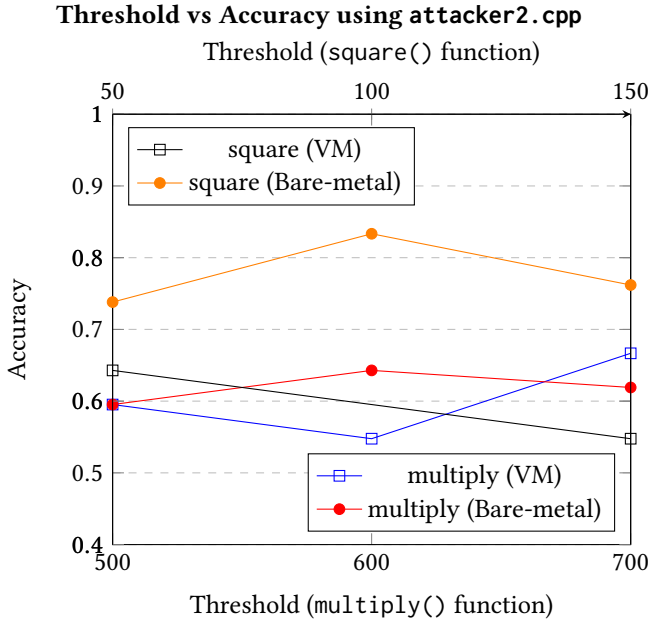

Figure 6. Threshold vs Accuracy using `attacker2.cpp` on the both setups. Notice the peak accuracy of the square() function on the bare-metal setup.

7 Countermeasures

There are a couple of possible countermeasure that can prevent this attack.

First, disabling shared memory from mapping to the same underlying physical address will indeed protect against this attack as it will eliminate the shared address through which the attacker can access the victim. However, implementing this will increase memory requirements for containers. Every container will have it's own shared libraries; shared libraries cannot be shared between containers to preserve space. This is not an ideal mitigation.

Another approach is to disable sharing cache between containers. However, this approach will necessarily require extended modification to containerization software and computer architecture that is running the containers. This is a software-based countermeasure, so it is more viable that other hardware based mitigation.

A more extreme approach is to only flush the LLC when a `clflush` is issued. Right now, calling `clflush` clears a cache line from all levels of cache [3]. However, if the `clflush` clears only the LLC, then the first and second level cache would still have the shared library address, thereby eliminating the large delta in access time; ultimately, eliminating the usability of this attack. This is a micro-architectural mitigation and therefore can only be delivered via kernel updates.

8 Source Code

All of the code of this project is stored in a Github repository, which can be found in https://github.com/sorliog/DockerFlushReload.

References

[1] Alexandru Agache, Marc Brooker, Andreea Florescu, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight Virtualization for Serverless Applications. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation* (Santa Clara, CA, USA) *(NSDI'20)*. USENIX Association, USA, 419–434.

[2] Intel Corporation. [n. d.]. 12th Generation Intel Core™ ® Processors Datasheet. https://www.intel.com/content/www/us/en/content-details/710723/12th-generation-intel-core-processors-datasheet-volume-2-of-2.html. , 43 pages. Accessed: May 2, 2023.

[3] Intel Corporation. [n. d.]. Combined Volume Set of Intel® 64 and IA-32 Architectures Software Developer's Manuals. https://cdrdv2.intel.com/v1/dl/getContent/671200. , 5-11 pages. Accessed: May 3, 2023.

[4] Intel Corporation. [n. d.]. Intel Core i34150 Processor 3M Cache 3.50 GHz Product Specifications. https://ark.intel.com/content/www/us/en/ark/products/77486/intel-core-i34150-processor-3m-cache-3-50-ghz.html. Accessed: May 2, 2023.

[5] Intel Corporation. [n. d.]. Intel Core i51245U Processor 12M Cache up to 4.40 GHz Product Specifications. https://ark.intel.com/content/www/us/en/ark/products/226260/intel-core-i51245u-processor-12m-cache-up-to-4-40-ghz.html. Accessed: May 2, 2023.

[6] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *Proceedings of the 23rd USENIX Conference on Security Symposium* (San Diego, CA) *(SEC'14)*. USENIX Association, USA, 719–732.